UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN
INSTITUTO DE INVERSIGACIONES EN MATEMÁTICAS
APLICADAS Y EN SISTEMAS

PROPUESTA DE PARALELIZACIÓN EN CLÚSTER DEL ALGORITMO SOM: DISEÑO DE GRANULARIDAD GRUESA

TESINA

QUE PARA OBTENER EL GRADO DE:

ESPECIALISTA EN CÓMPUTO DE ALTO RENDIMIENTO

PRESENTA
DAVID DANIEL ANGUIANO ARÉVALO

TUTOR
DR. PAUL ERICK MÉNDEZ MONROY

CIUDAD UNIVERSITARIA, CIUDAD DE MÉXICO, MARZO 2024

Agradecimiento

Agradezco a Dios y a mis padres por haberme apoyado a culminar mis estudios de especialización, he tenido la fortuna de tener una familia como la que tengo, es necesario tener que decir que he pasado por diferentes vicisitudes en la vida, desde tener problemas de salud graves hasta la perdida de mi padre, sin embargo, he podido levantarme por el apoyo que he recibido de ellos. No puedo olvidar también la grandiosa fortuna de tener un tutor de tesina que desde sus posibilidades ha estado siempre ayudándome a poder concluir con mi tesina, no sabe cuánto se lo agradezco. A los miembros del jurado el Dr. Óscar Alejandro Esquivel Flores y el Dr. Erik Molino Minero Re.

A la Universidad Nacional Autónoma de México por darme la oportunidad de cursar una educación de excelencia de manera gratuita y cumplir uno más de mis sueños. Sumado al agradecimiento por el apoyo parcial de los proyectos PAPIIT-DGAPA IA102620 y IN105623.

Índice general

Índice de figuras

1. Introducción

Un mapa auto-organizado (*SOM, Self-Organizing Map*) es un algoritmo de aprendizaje no supervisado con el objetivo de visualizar datos de alta dimensión y agrupamiento principalmente, en su forma básica produce un grafo de similitud de los datos de entrada. Convierte las relaciones estadísticas no lineales entre los datos de alta dimensión en relaciones geométricas simples de dimensiones reducidas, generalmente una red o rejilla regular de nodos llamados "*neuronas"* de una o dos dimensiones. Como la *SOM* comprime la información conservando las relaciones topológicas y/o geométricas más importantes de los datos, también se puede pensar que produce algún tipo de abstracción de los datos [1]. Estos dos aspectos, visualización y abstracción, se pueden utilizar de diversas formas en tareas complejas como el análisis de procesos, percepción de máquina, control, y comunicación.

En su forma básica la *SOM* fue concebida por el Dr. Kohonen en 1982 [1] y en la actualidad, se han publicado alrededor de 40,000 artículos de investigación que involucran a la *SOM* como parte principal de su aplicación [2]. Muchos textos y tutoriales sobre *SOM* han aparecido, teniendo implementaciones dentro de los principales lenguajes de programación como Java, C, C++, R, Python, Matlab y Ruby. Además, varios paquetes de software contienen todos los procedimientos principales, programas de monitorización, diagnóstico y visualización, y también hay datos de ejemplo disponibles gratuitamente en Internet.

El algoritmo *SOM* requiere de dos etapas para su implementación, la etapa de entrenamiento y la etapa de inferencia [3]. La etapa de entrenamiento que se encarga de ajustar las neuronas (actualizar) dentro de la rejilla en función de todos los vectores de entrada para asimilar su topología. La etapa de inferencia realiza la selección de la neurona con mayor similitud ante un nuevo vector de entrada con el fin de visualizar en que parte de la rejilla se encuentra y asignarle una posición en la topología.

Sin embargo, con el crecimiento en la adquisición de información a gran escala y de alta dimensión, la aplicación del algoritmo *SOM* aumenta su tiempo en la etapa de entrenamiento cuando se emplea el algoritmo clásico secuencial, lo que limita realizar su análisis para agrupamiento, reducción de dimensión y posibilidad de funcionar como clasificador. Esto presenta un área de investigación para realizar su implementación a través de estrategias de cómputo de alto rendimiento.

1.1. Problema

La utilización del algoritmo de *SOM* secuencial para la visualización de gran cantidad de datos de alta dimensión requiere de una cantidad considerable de tiempo de entrenamiento o en algunos casos es inviable. Esto debido a que el entrenamiento actualiza la matriz de pesos cada vector de entrada, lo que permite preservar la topologia de los datos, pero aumentando el tiempo de entrenamiento cuando aumentan los datos de entrada.

dificulta su desarrollo y aplicación aún cuando tiene injerencia en muchas áreas de investigación.

1.2. Objetivo

Diseñar una versión paralela para el entrenamiento de una red neuronal de mapa auto-organizado tipo *SOM* preservando la topología de los datos de entrada, para una gran cantidad de datos.

1.3. Objetivos específicos

- Investigar y optimizar los procedimientos que limitan la ejecución eficiente del algoritmo *SOM* en paralelo.
- Diseñar una versión *SOM* paralela dentro de un lenguaje de medio nivel como *MPI*.

En este trabajo se describe una propuesta de una versión paralela del algoritmo de mapas autorganizados con aprendizaje no supervisado. Desde la incorporación de un conjunto de datos para las pruebas, el análisis de los procedimientos con un alto costo computacional a través del diseño de su versión serial en lenguaje *C*.

2. Antecedentes

En la siguiente sección se describe el algoritmo de *SOM*, su entrenamiento en forma serial y la inferencia. Se presenta en detalle cada uno de los cuatro pasos principales para el entrenamiento (cálculo de la distancia de un dato a todas las neuronas, la neurona ganadora, la vecindad de influencia y el ajuste del mapa autoorganizado), donde el mapa de neuronas final permite inferir nuevos datos y obtener una posible clasificación.

Además de la descripción del algoritmo *SOM* con el objetivo de paralelización, se presentan los elementos de paralelización distribuida mediante la librería *MPI* (*Message Passing Interface*), definiendo la configuración de los nodos dentro de una red, los elementos para la comunicación de datos entre ellos y la sincronización para evitar fallas en los cálculos como la condición de carrera.

2.1. MAPAS AUTO-ORGANIZADOS

La *SOM* puede ser descrita formalmente como un mapeo no lineal, ordenado y suave de datos con alta dimensión a elementos de una matriz regular de baja dimensión. Tomando como vectores de referencia las llamadas "*neuronas*" que son vectores ordenados y distribuidos en el espacio de entrada vectorial, cuyo ajuste define un tipo de "*red elástica*" llamada normalmente como rejilla o mapa autoorganizado que caracteriza el orden topológico de este mapeo o distribución.

El proceso en el que se forman tales adaptaciones de las neuronas está definido por un aprendizaje competitivo[4], donde diversos pasos del algoritmo clásico generan cuellos de botella para su aplicación con gran cantidad de datos. A continuación, se presenta la etapa de entrenamiento Serial del algoritmo para ajustar los pesos (neuronas) del mapa autoorganizado actualizándolo con cada dato de entrada, en un proceso repetitivo hasta cumplir alguna condición de paro.

2.1.1. Etapa de entrenamiento SOM serial

En general, una red *SOM* se entrena utilizando un aprendizaje no supervisado que produce una representación neuronal en un espacio con reducción de dimensiones con respecto al conjunto de entrada para entrenamiento, con el objetivo general de preservar las propiedades de topología del espacio de entrada [5].

El objetivo del mapeo es representar todas las muestras del conjunto de entrada X por un conjunto de vectores de peso llamados neuronas W de tal manera que las muestras muy cercanas en el espacio de entrada estén representadas por neuronas que también están muy cerca. La Figura 2.1 ilustra un ejemplo esquemático de un *SOM* en 2D.

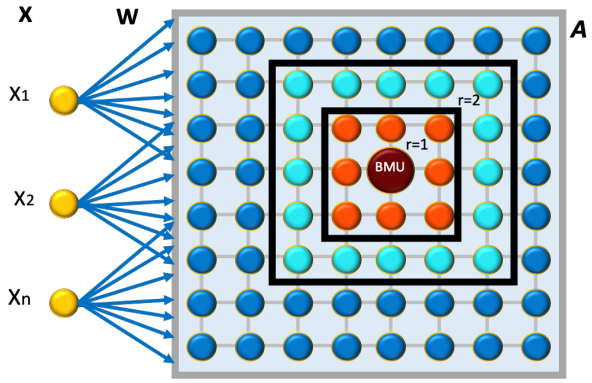

Figura 2.1 Una estructura rectangular SOM, con BMU y vecindades r=1,2.

El mapeo utiliza una rejilla A (lineal, rectangular, hexagonal, etc.) donde cada vector de pesos w_k es asignado a cada neurona dentro de la rejilla $k \in A$. Este mapeo *SOM* $M_A: X \to A$ tiene un entrenamiento competitivo donde los pesos de las neuronas se actualizan en función del conjunto de entrenamiento de entrada, esta fase de entrenamiento se repite un cierto número de veces llamadas épocas E_p. Una descripción de las tareas principales del entrenamiento secuencial de una *SOM* se muestra en el diagrama de flujo de la Figura 2.2.

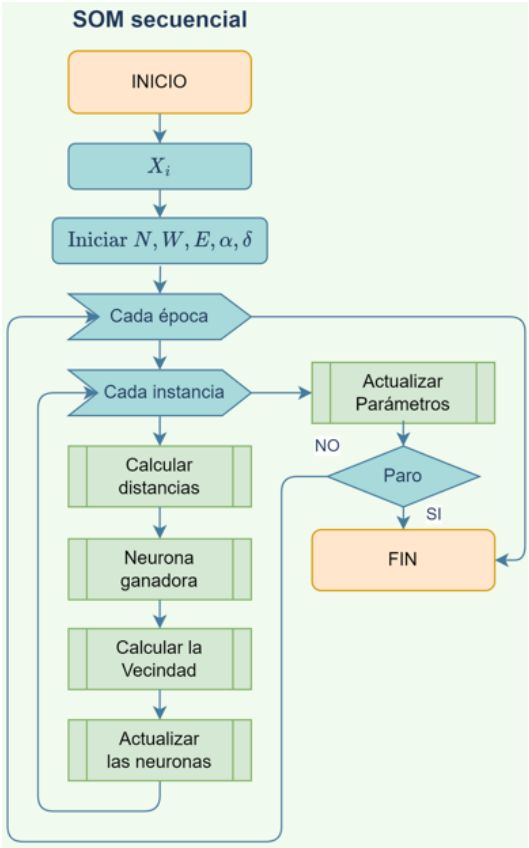

Figura 2.2 Diagrama de flujo Secuencial de la *SOM.*

En la Figura 2.3 se describe en pseudocódigo del entrenamiento en serie de una *SOM* con rejilla rectangular 2D, los detalles se describen a continuación:

La entrada es un conjunto de entrenamiento $X = \{x_1, x_2, \dots, x_T\} \in R^{Txn}$ de los vectores de entrada $x_T \in R^n$ con T muestras y la salida es una matriz de pesos $W = \{w_k | k \in A\} \in R^{Axn}$ entrenada. Los parámetros de diseño contemplan la propuesta de una red A predefinida normalmente rectangular o hexagonal, una función de vecindad h, épocas de repetición E_p, el factor de aprendizaje α y el radio de vecindad δ.

Figura 2.3 Algoritmo SOM Serial

Entrada:
Los datos de entrenamiento $X = \{x_1, x_2, \dots, x_T\} \in R^{Txn}$ con T muestras de n dimensiones.
Salida:
La matriz de pesos entrenada
Setup:
1. Establecer A, h_{ck}, α, δ y E_p.
Entrenamiento serial:
1. Inicializando $W = \{k \in A\}$
2. For $e_p = 1 \dots E_p$
3. For $t = 1 \dots T$
4. $d \leftarrow$ Las distancias computarizadas: $(x_t, w_k)\ \forall\ k \in A$ (2.1)
5. $c \leftarrow$ Encontrar la BMU (2.2)
6. $h_{ck} \leftarrow$ Determinar el conjunto de vecindad por la BMU (2.3)
7. $W \leftarrow$ Actualizar la BMU y sus vecindades (2.4)
8. end t
9. Ajustar α y δ.
10. end e_p

El entrenamiento comienza inicializando la matriz de pesos W, que contiene las neuronas (vectores de pesos) $w_k \in R^n$, los pesos se pueden inicializar como valores aleatorios, extrayendo muestras aleatorias de X o a través de un análisis de componentes principales (*PCA, Principal Component Analysis*), entre otros [5]. La parte principal del algoritmo son los pasos 4 al 7, que se repiten para cada muestra del conjunto de entrenamiento $t = 1 \dots T$.

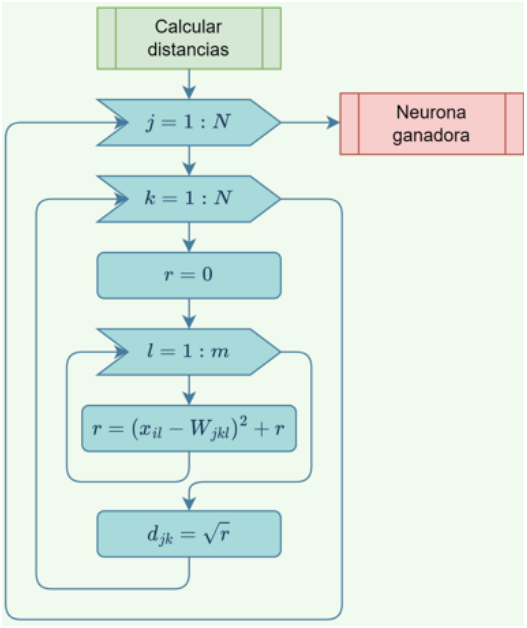

Figura 2.4 Diagrama de flujo para cálculo de la distancia de un vector de entrada.

Primero en el paso 4, con el vector de entrada actual x_t se calculan distancias euclidianas d_k (eq. 2.1) a todos los vectores de pesos w_k (Figura 2.4).

$$d_k = \left|\left|x_t - w_k\right|\right|^2 \tag{2.1}$$

El quinto paso involucra el cálculo de una neurona ganadora *BMU* (*Best Match Unit)* (eq. 2.2) seleccionando la neurona con el argumento mínimo en d_k, la neurona ganadora (Figura 2.5) es un componente clave en el aprendizaje competitivo entre las neuronas.

$$BMU = c = arg\ arg\ min\ (d_k) \tag{2.2}$$

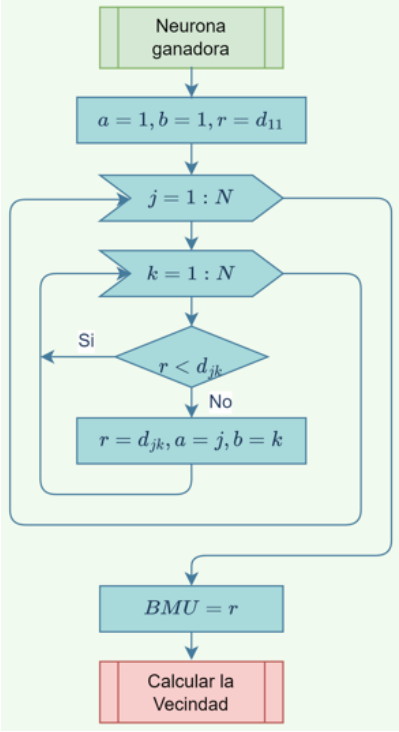

Figura 2.5 Diagrama de flujo para calcular la neurona ganadora.

Para actualizar la matriz de peso W es necesario seleccionar las neuronas vecinas alrededor de la ganadora, un conjunto de vecindad h_{ck} (eq. 2.3) donde todas las neuronas determinan su vecindad de acuerdo con una distancia neural de radio r, que disminuye rápidamente para las neuronas alejadas de la neurona ganadora en las coordenadas de una cuadrícula, a menudo las regiones vecinas se consideran gaussianas:

$$h_{ck} = exp\left(-\frac{||r_c - r_k||^2}{\delta(t)^2}\right) \quad (2.3)$$

donde r_c y r_k son las coordenadas de las neuronas ganadora (c) y su vecina (k) en el mapa *SOM*. El ancho $\delta(t)$ de la función de vecindad disminuye monotónicamente en base al índice de época e_p durante el entrenamiento, desde un valor inicial comparable a la dimensión de la red, hasta un valor final efectivamente igual al ancho de una neurona individual.

La actualización de la matriz de pesos utiliza un entrenamiento competitivo, esta actualización de la *BMU* y su conjunto de vecindad de acuerdo con la regla de aprendizaje (eq. 2.4) (Figura 2.6). La idea es que esas neuronas cercanas a la *BMU* obtendrán parte de la información proporcionada por la muestra actual x_t.

$$w_k = w_k + h_{ck}\, \alpha\, (x_t - w_k) \quad (2.4)$$

Finalmente, después de que todos los vectores de entrada se usen para actualizar la matriz de pesos se requiere aumentar el índice de época e_p. Además, se ajustan la velocidad de aprendizaje α y el radio de vecindad δ que disminuyen monotónicamente con e_p, mientras que la época máxima no se alcance $e_p < E_p$ (Figura 2.7).

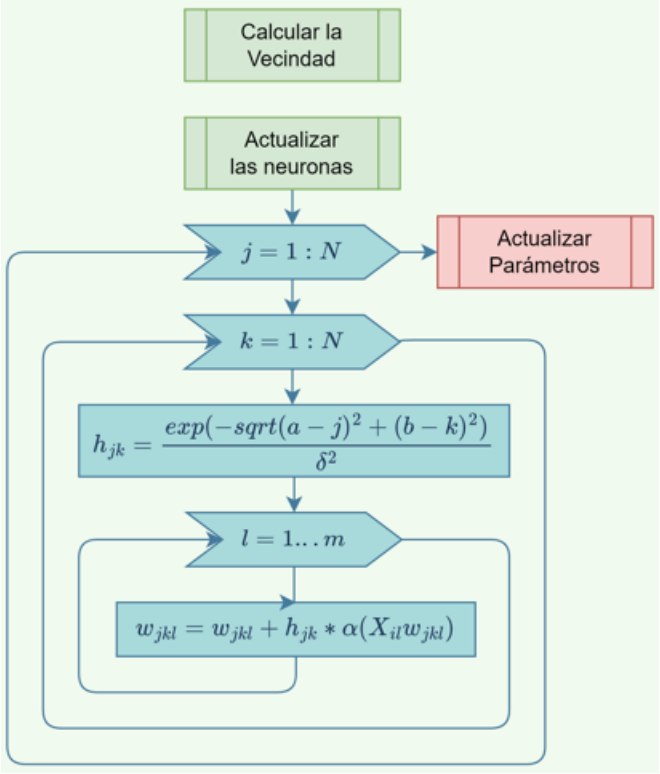

Figura 2.6 Diagrama de flujo para el cálculo de la vecindad y actualización de neuronas.

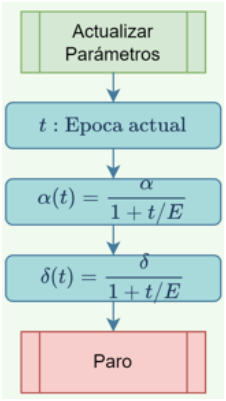

Figura 2.7 Diagrama de flujo para actualizar α(*t*) y δ(*t*) en cada época *t*.

2.2. Paralelización distribuida

Interfaz de paso de mensaje *MPI* [6] es un estándar que define la sintaxis y la semántica de las funciones contenidas en una biblioteca de *C*, *C++*, *Python*, etc., diseñado para usarse en programas que exploten la existencia de múltiples computadoras. El paso de mensajes es una técnica empleada en programación concurrente para aportar sincronización entre procesos y permitir la exclusión mutua. Su principal característica es que no precisa de memoria compartida, por lo que es muy importante en la programación de sistemas distribuidos.

El esfuerzo para estandarizar *MPI* involucró a cerca de 60 personas de 40 organizaciones diferentes principalmente de EE. UU. y Europa. La mayoría de los vendedores de computadoras concurrentes estaban involucrados con *MPI*, así como con investigadores de diferentes universidades, laboratorios del gobierno e industrias.

2.2.1. Configuración de nodos

Con *MPI* el número de procesos requeridos se asigna antes de la ejecución del programa, y no se crean procesos adicionales mientras la aplicación se ejecuta. A cada proceso se le asigna una variable que se denomina rango, la cual identifica a cada proceso, en el rango de 0 a $p - 1$, donde p es el número total de procesos [7].

En *MPI* se define un comunicador como una colección de procesos, los cuales pueden enviarse mensajes el uno al otro; el comunicador básico se denomina `MPI_COMM_WORLD` y se define mediante un macro del lenguaje C, agrupando a todos los procesos activos durante la ejecución de una aplicación.

Las llamadas de *MPI* se dividen en cuatro clases: configuración de nodos y comunicación, comunicación punto a punto, comunicación colectiva y configuración de datos definidos por el usuario. Los tipos de datos tanto de envío como de recepción deben ser iguales. [8]

MPI disponen de cuatro funciones primordiales que se utilizan en todo programa con *MPI*:

Inicialización: `int MPI_Init ( int *argc , char ***argv )`. Inicializa el entorno *MPI*

Identificación del proceso: `MPI_Comm_rank ( MPI_Comm comm , int *rank)`. El proceso obtiene el rango asignado dentro del comunicador *comm* que será su identificador.

Procesos en el comunicador: `MPI_Comm_size ( MPI_Comm comm , int *size)`. Obtiene el total de procesos dentro de un comunicador *comm*.

Finalización: `int MPI_Finalize ( )`. Finaliza el entorno *MPI* acabando el programa.

2.2.2. Comunicación entre procesos y sincronización

Comunicación punto a punto. La transferencia de datos entre dos procesos se consigue mediante las llamadas `MPI_Send` y `MPI_Recv` y sus variantes. Además de estas otras llamas importantes se emplean para poner barreras y realizar verificación. Estas llamadas devuelven un código que indica su éxito o fracaso de tipo entero.

Envío: `int MPI_Send(*data, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm)`. Permite enviar *data* al proceso destino *dest*

Recepción: `Int MPI_Recv(void *data, int count, MPI_Datatype type, int source, int tag, MPI_Comm comm, MPI_Status *st)`. Espera recibir *data* desde el proceso fuente *source* revisando su estado de recepción en *st*.

Bloqueante: `int MPI_Wait (MPI_Request *req, MPI_Status *st)`. Bloquea el proceso hasta que termine la operación de envío o recepción en llamadas no bloqueantes.

Comprobar envío: `int MPI_Test(MPI_Request *req, int *flag, MPI_Status`

`*st)`. Comprueba si una operación específica de envío o recepción se ha completado sin bloquear el proceso.

Llamada sincronizante: `int MPI_Barrier(MPI_Comm comm)`. Realiza operaciones de sincronización sin intercambio de información, para continuar la ejecución de manera concurrente de todos los procesos.

Comunicación colectiva. Para la transferencia de datos entre más de un proceso de manera eficiente, *MPI* contempla un conjunto de funciones con operaciones como difusión (*broadcast*), distribución (*scatter*) y recolección (*gather*) de datos, por mencionar las más importantes. A continuación, se enlistan las empleadas durante el desarrollo de este proyecto.

Difusión: `int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int source, MPI_Comm comm)`, envía un único mensaje desde el proceso que tenga el rango *source* a todos los procesos del grupo definidos en el comunicador *comm*, incluyendo a sí mismo. Equivalente a ejecutar una comunicación punto a punto para todos los procesos en el *comm*.

Distribución: `int MPI_Scatter(void *sendbuf, int scount, MPI_Datatype stype, void *recvbuf, int rcount, MPI_Datatype rtype, int source, MPI_Comm comm)`. El proceso source divide los datos contenidos en sendbuf en size partes para enviarlos de manera equitativa a todos los procesos dentro del comunicador comm, cada proceso almacena su parte en su variable local recvbuf.

Recolección: `int MPI_Gather(void *sendbuf, int scount, MPI_Datatype stype, void *recvbuf, int rcount, MPI_Datatype rtype, int dest, MPI_Comm comm)`. Recoge una serie de datos de tamaño scount desde los procesos definidos en el comm en un único proceso destino dest, almacenándolo en la variable recvbuf en el orden del rango de cada proceso.

Definir tipos de datos por el usuario. Si se tiene la necesidad del envío de datos de tipo diferente a los definidos en el estándar de *MPI*, se pueden definir tipos de datos adoc por el usuario, estos tipos de datos derivados están permitidos para el parámetro *datatype*. La concordancia en tamaño entre los tipos de datos especificados por los procesos receptores y del emisor debe ser igual. Esto implica que la cantidad de datos enviados debe ser la misma cantidad que los datos recibidos entre cada proceso con respecto al proceso fuente. [7]

Para definir nuevos tipos de datos se puede utilizar la llamada *MPI_Type_struct* para crear un nuevo tipo o se puede utilizar la llamada *MPI_Pack* para empaquetar los datos.

Empaquetado: `int MPI_Pack(void *inbuf, int incount, MPI_Datatype datatype, void *outbuf, int outsize, int *position, MPI_Comm comm)`

Estructura: `int MPI_Type_struct(int count, int *array_of_blocklengths, MPI_A int *array_of_displacements, MPI_Datatype *array_of_types, MPI_Datatype *newtype)`

3. Trabajo Relacionado

En la actualidad la necesidad cuando no se tienen datos etiquetados es agrupar un conjunto de datos masivo para su posible etiquetado y a su vez conservar la topología original, esto permite una visualización de los grupos y su extracción. En este trabajo una paralelización de los mapas autoorganizados es un requerimiento fundamental para la implementación y usabilidad de la red con gran cantidad de datos.

Además del uso de la *SOM* en su forma básica existen centenar de variantes que emplean el agrupamiento por *SOM*, por ejemplo: Kaski et al. [5] y Oja et al. [9], el Mapeo de Topografía Generativa *GTM* (*Generative Topography Mapping*) por Bishop y Svensen [10], también el estudio realizado por Pena y Barbakh [13] sobre mapas topográficos y visualización de datos, mostrando una comparación exhaustiva entre *SOM*, entre que los que destacan la red Gas Neural Creciente *GNG* (*Growing Neural Gas*), que también es capaz de coincidir con la distribución temporal del conjunto de datos original [11], pero es demasiado costoso computacionalmente.

3.1. Paralelizaciones recientes

Los algoritmos disponibles en la literatura para paralelizar *SOM* se pueden agrupar en dos categorías principales: (a) partición de red *SOM* y (b) partición de datos de entrada.

Además, los mapas autoorganizados paralelos se pueden clasificar en: (1) paralelización de la *SOM* original, conocido también como *SOM* en línea, implementado principalmente en hardware ad-hoc [12] y (2) paralelización de una aproximación de la *SOM* original, es decir, una variante del algoritmo *SOM*, implementado en sistemas de computación distribuida con la desventaja de preservar en menor medida la topología del conjunto de datos [13].

Jin et al. [14,15] presentaron una paralelización llamada *ESOM*, para preservar mejor la topología del conjunto de datos original y para resolver problemas de optimización complejos; (2) la *SOM* de crecimiento (*GSOM*) para lograr la misma precisión topológica de los métodos *GTM*, ampliamente estudiado en esta década (por ejemplo, [16-18]); y (3) el SOM de densidad relativa (*ReDSOM*) propuesto recientemente por [19] para modelar conjuntos de datos de alta densidad con topología variable, que puedan alcanzar el mismo rendimiento del *GNG*.

Gajdoš y Moravec [21] propone una modificación simple del algoritmo *SOM* particionando los datos de cada entrada y también la rejilla, preseleccionando los centroides potenciales y usándolos como vectores para una red *SOM* final, esto lo aplica al reconocimiento de dígitos escritos a mano obteniendo un mapa U-matriz similar a su versión *SOM* secuencial empleando la unidad de procesamiento de gráficos *GPU* (*Graphics Processing Unit*).

Uno de esos casos de trabajo fue realizado por Atanas Radenski [23] donde midió el rendimiento del paralelo ordenar en un clúster híbrido. Luego midió el rendimiento en un clúster *MPI* puro y de manera similar en un *OpenMP* (*Open Multi-Processing*) [22] entorno. Comparó todas estas diferentes implementaciones basadas en parámetros como el número de

Subprocesos *OpenMP*, procesos *MPI*, nodos y núcleos utilizados. Llegó a la conclusión de que cuando todo el conjunto era lo suficientemente pequeño para encajar en la RAM, la versión *OpenMP* del algoritmo mostró mejores resultados que la implementación de *MPI,* mientras que el rendimiento mostrado por el algoritmo híbrido *OpenMP – MPI* disminuyó con respecto a sus versiones puras de *OpenMP* y MPI implementaciones.

Faro, Giordano y Maiorana [20] proponen tres algoritmos novedosos de clúster paralelo para la *SOM* de Kohonen. En todos estos algoritmos los datos a ser agrupados se subdividen entre los nodos de una *GRID*. En los dos primeros algoritmos cada nodo ejecuta un *SOM* en línea, mientras que en el tercer algoritmo los nodos ejecutan una *SOM* hibrida línea-lotes llamado *MANTRA*. Los algoritmos difieren en cómo los pesos calculados por los nodos trabajadores son recombinados por un maestro para lanzar la próxima época del *SOM* en los nodos. Un esquema de prueba empleando demuestra la convergencia de las *SOM* paralelas propuestos, obteniendo grupos significativos en un conjunto de datos con mejoras en el tiempo de entrenamiento empleando *MPI*.

Patel et al. [24] propone un algoritmo paralelo en un Clúster con *MPI-OpenMP* para reducir el tiempo de entrenamiento y mejorar el rendimiento de la *SOM*. Realizando pruebas con el algoritmo serial y por lotes de la *SOM* a través de paralelizaciones *MPI*, *OpenMP* y *MPI-OpenMP*. Los resultados del algoritmo demostraron una aceleración de hasta 15.3% en comparación con el entrenamiento secuencial de la *SOM*.

El presente trabajo propone una modificación simple para la paralelización del algoritmo secuencial *SOM* implementado en *MPI* con el objetivo de preservar la topología de los datos y mejorar el tiempo de entrenamiento en grandes volúmenes de datos.

4. Metodología

La presente metodología supone una paralelización del algoritmo *SOM* serial en una forma distribuida a través de la librería *MPI,* en una arquitectura Maestro-Trabajadores empleando comunicación colectiva. El nodo Maestro se encarga de distribuir los parámetros necesarios para el algoritmo *SOM* y las matrices iniciales, así como establecer la sección de los datos que los nodos Trabajadores necesitan para realizar el entrenamiento de su *SOM* local.

Una vez que todos los nodos Trabajadores realicen su entrenamiento cumpliendo los criterios de paro. El nodo Maestro recibirá cada una de las matrices *SOM* locales desde los nodos Trabajadores. Como paso final, se realiza en el nodo Maestro un entrenamiento *SOM* serial con las neuronas de las *SOM* locales como entradas, obteniendo un mapa llamado *SOM* global.

A continuación, se describen los pasos principales para la implementación del algoritmo *SOM* paralelo, así como los procesos de comunicación involucrados en la distribución y recolección de parámetros y matrices, estos se muestran en dos diagramas de flujo, uno para el nodo Maestro en la Figura 4.1 y otro para los nodos Trabajadores Figura 4.2.

4.1. Nodo maestro

1. Se configuran todos los parámetros y datos para el entrenamiento, tanto para el nodo Maestro como los Trabajadores. El nodo maestro lee las muestras de entrada X, luego Inicia las matrices de pesos W_s, inicializa las épocas E que realiza cada uno de los trabajadores, la época global para el entrenamiento final E_G, la tasa de aprendizaje alfa α y la anchura de la vecindad delta δ. Para garantizar que el Maestro termina de configurar los parámetros de las *SOMs* se aplica una barrera (*MPI barrier 1*) a todos los nodos.
2. Se envían las dimensiones de los datos y el mapa, mediante una operación *MPI Broadcast* el Maestro envía los parámetros (N, m, T) que son la dimensión de la matriz de pesos W_s, la dimensión de las muestras y el total de muestras, respectivamente. Con estos parámetros los trabajadores conocen y asignan la memoria para su matriz de pesos y las muestras que le tocan procesar.
3. El nodo Maestro entonces envía la matriz de pesos W_s de (NxN) a los Trabajadores a través de un *MPI broadcast*.
4. Después el Maestro envía los parámetros (E, δ, α) que son la época de entrenamiento E y la anchura de la vecindad δ y la tasa de aprendizaje α. Parámetros necesarios para el entrenamiento *SOM* de los S trabajadores.

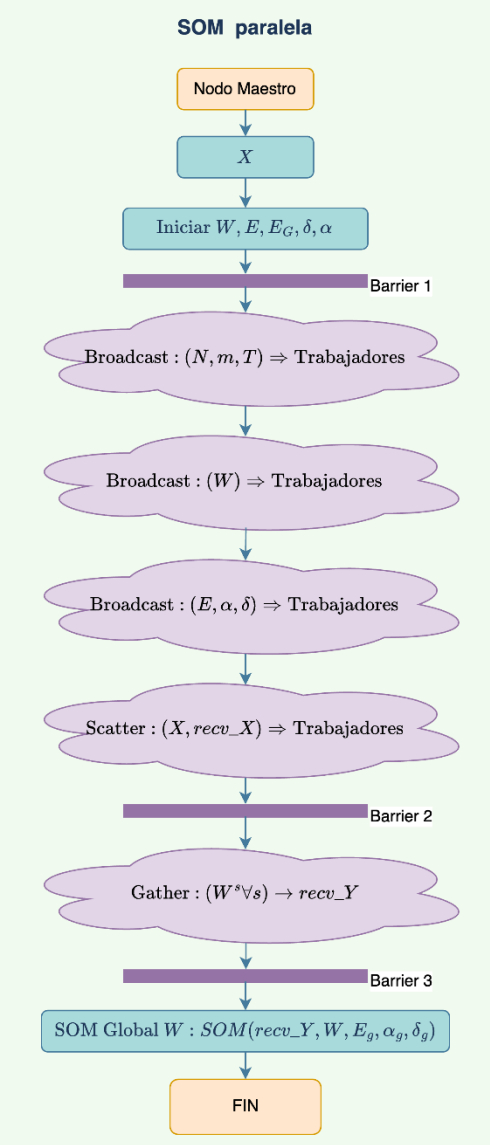

Figura 4.1 Diagrama de flujo del nodo Maestro del entrenamiento SOM distribuido.

5. Con un operador *MPI scatter* el Maestro reparte las muestras particionando X para que los trabajadores reciban su parte en $recv\_X$, que es un arreglo local. Con estas muestras cada uno de los trabajadores realizará su entrenamiento. Para garantizar que todos los trabajadores reciban los parámetros y la parte de las muestras antes de empezar a entrenar se aplica una barrera *MPI barrier* 2.
6. Aquí inicia el trabajo de los Trabajadores. Ya que recibieron todos los parámetros y su parte de las muestras para realizar sus *SOM* locales, se procede a realizar el entrenamiento de manera serial descrito en la Figura 4.2.
7. Durante su entrenamiento el Maestro espera para recopilar las SOM locales de cada uno de los trabajadores. Mediante un *MPI gather* se reciben las SOMs locales de todos los trabajadores. Todas las *SOMs* locales las guarda en $recv\_Y$ que es un arreglo de tamaño igual a $NxNxS$ para guardar todas las matrices W_s de los S trabajadores.

8. Se coloca una última barrera para garantizar que se reciben todas las matrices de neuronas W_s y realizar la *SOM* global
9. Finalmente, el Maestro entrena la matriz de pesos final W usando cada una de las neuronas de las *SOMs* locales como entradas al algoritmo y los parámetros de entrenamiento: época global E_G, tasa de aprendizaje α y anchura de la vecindad δ.

4.2. Nodo trabajador

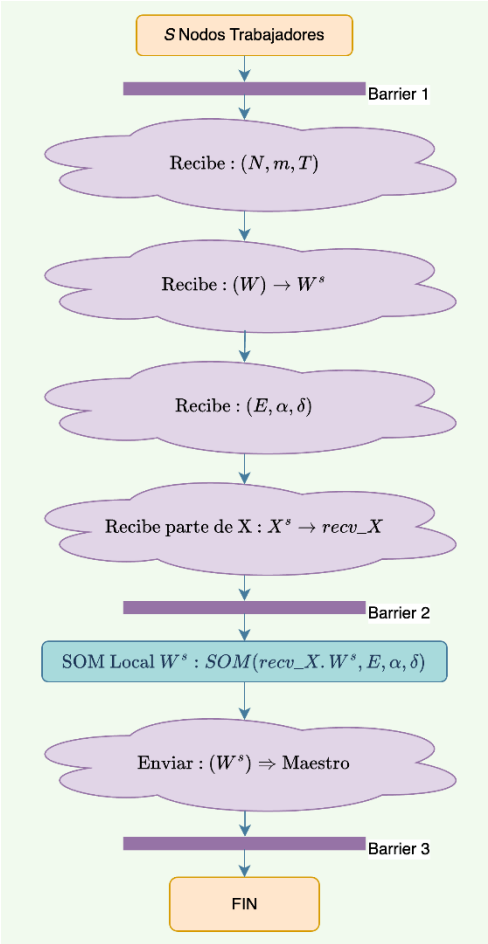

Figura 4.2 Diagrama de flujo de un nodo Trabajador del entrenamiento *SOM* local.

1. Los trabajadores *S* se encargarán de realizar el entrenamiento de su *SOM* local. Estos inician cuando el Maestro envía los parámetros y muestras para configurar la *SOM* local. Con la primera barrera (*MPI barrier 1*) se garantiza que el Maestro termina de configurar los parámetros del algoritmo.
2. Los trabajadores reciben del nodo Maestro usando un *MPI broadcast* los parámetros (N, m, T) que son: la dimensión de la matriz de pesos W_s, la dimensión de las muestras y el total de muestras.
3. Después cada trabajador a través de un *MPI broadcast* recibe la matriz de pesos inicial y la guarda en W_s.

4. Para realizar su propio entrenamiento a través de un *mpi broadcast* se recibe la época local E y la anchura de la vecindad δ además de la tasa de aprendizaje α.
5. Finalmente recibe la sección de las muestras a través de un *MPI scatter*, para realizar el entrenamiento de la *SOM* W_s con estas muestras X_s. Una segunda barrera (*MPI barrier 2*) garantiza que todos los trabajadores reciban los parámetros y la parte de sus muestras a trabajar antes de iniciar su entrenamiento.
6. Cada trabajador realiza el entrenamiento de su propia *SOM* W_s un determinado número de épocas de entrenamiento E, mientras el Maestro espera a que todos los trabajadores terminen.
7. Una vez que el trabajador termina su entrenamiento, envía su *SOM* local al Maestro mediante un *MPI gather* y el Maestro la guarda en parte de vector $recv\_Y$.
8. El trabajador termina así su tarea. Todos los trabajadores destruyen sus variables y conexiones (*MPI finalize*) una vez concretado el envío de su *SOM*.

5. Implementación y resultados

La implementación se realizó en el clúster Lucar del Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas de la UNAM. Realizado en 4 nodos asignados, este cuenta con un sistema de archivos distribuido que permite compartir archivos de manera transparente.

Se empleo el conjunto de datos de *MNIST(Modified National Institute of Standards and Technology database)* que contiene imágenes de 28x28 pixeles con los números del 0 al 9 en forma escrita de diversos participantes. El objetivo de aplicar el algoritmo SOM es buscar el agrupamiento.

5.1. Entrenamiento serial

Se realizó el entrenamiento variando los parámetros como α y δ, el número de neuronas, las épocas buscando el mejor rendimiento en base a dos índices de desempeño, el primero es el error topológico que establece que tan cerca esta una neurona ganadora de su segunda neurona, un valor cercano a cero del índice determina un mayor agrupamiento de neuronas similares (formando grupos). El segundo es el error de cuantización que establece que tan bien representa la neurona ganadora de una muestra determinada, entre más bajo este valor, significa que se tiene una mejor representación de las muestras dentro del mapa.

Primero se entrena la red variando la tasa de aprendizaje alfa inicial en el rango de [0.7:0.05:0.99], asegurándose que la anchura inicial de la vecindad se mantenga en el valor de 1, se realiza la ejecución de 25 épocas con 12x12 neuronas. Se cálculo el error de cuantización (QE) y el error topológico (TE) dando como resultado la Figura 5.1. Seleccionando el menor valor de la sumatoria de QE y TE, que fue para $\alpha = 0.75$ y $\delta = 1$, así $\alpha = 0.75$ se utilizó para para encontrar el mejor valor de δ.

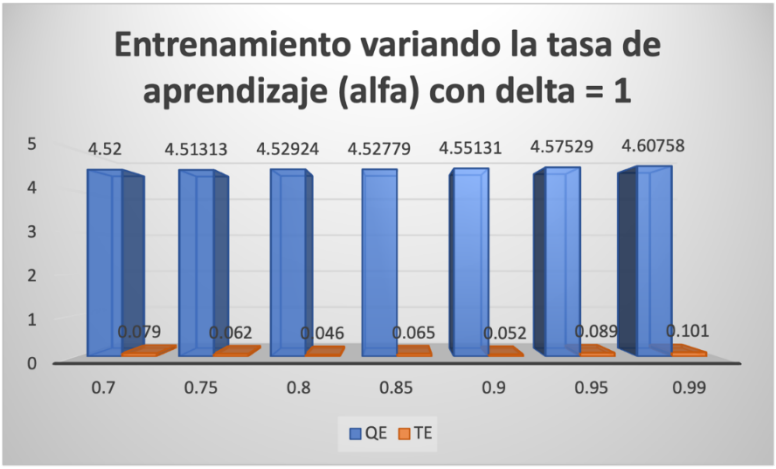

Figura 5.1 Entrenamiento SOM serial variando el parámetro α.

De las ejecuciones anteriores se definió que el parámetro $\alpha = 0.75$ tiene la mejor relación entre el error topológico y el de cuantización, con este parámetro se varió el

parámetro δ en el rango de [0.7:0.05:1], asegurándose que la tasa de aprendizaje $\alpha = 0.75$, se realiza la ejecución de 25 épocas con 12x12 neuronas. El *QE* y *TE* para el rango es mostrado en la Figura 5.2, de esta se seleccionó la mejor relación *QE* y *TE*, que fue para $\alpha = 0.75$ y $\delta = 0.95$, que es utilizado para definir el mejor tamaño de la red con diferentes números de épocas de entrenamiento.

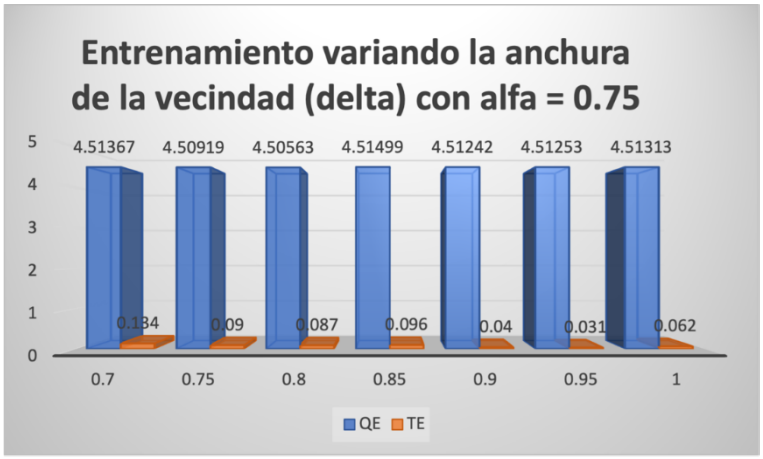

Figura 5.2 Entrenamiento SOM serial variando el parámetro δ.

Con los parámetros $\alpha = 0.75$ y $\delta = 0.95$ fijos, se realiza un entrenamiento variando las épocas [5:5:50] con 10 neuronas, en la Figura 5.3 se muestra el comportamiento del *QE* y *TE* donde se observa que 30 épocas de entrenamiento obtienen los mejores valores sin requerir un mayor tiempo de entrenamiento ya que la mejora no es considerable.

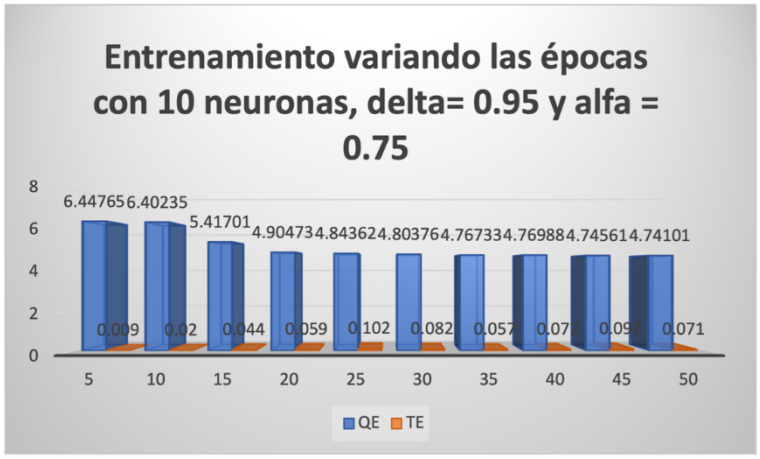

Figura 5.3 Entrenamiento SOM serial variando las épocas con 10x10 neuronas.

Para conocer un número adecuado de neuronas para la red se modificaron por 12x12 y 14x14 neuronas con los parámetros α y δ fijos y variando el número de épocas de entrenamiento. Su desempeño en función del QE y TE se muestran en las Figuras 5.4 y 5.5, mostrando que para ambos casos el número de épocas adecuado es de 30, ya que un número mayor no mejora el desempeño.

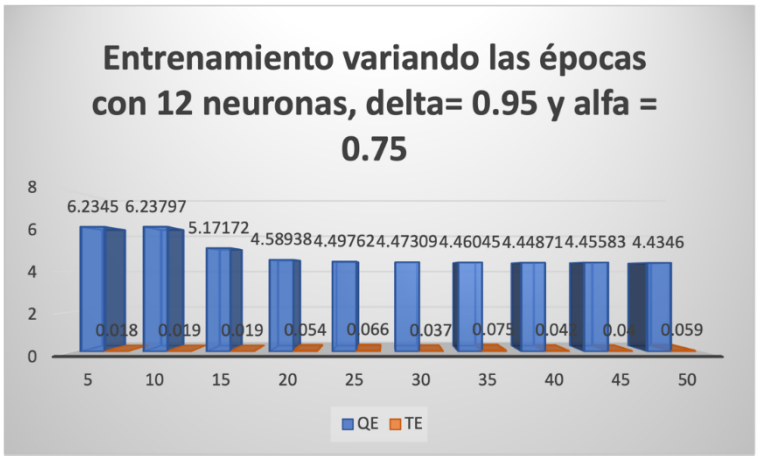

Figura 5.4 Entrenamiento SOM serial variando las épocas con 12x12 neuronas.

De la Figura 5.5 se observa que tiene la mejor relación $QE = 4.2$ y $TE = 0.061$, lo que muestra una distancia promedio de 0.0042 entre una imagen dígito 28x28 normalizada y su neurona ganadora dentro de la red y que sólo los 61 de los 1000 de las imágenes dígito no tiene a la segunda neurona ganadora como vecina, lo que muestra una preservación de la topología.

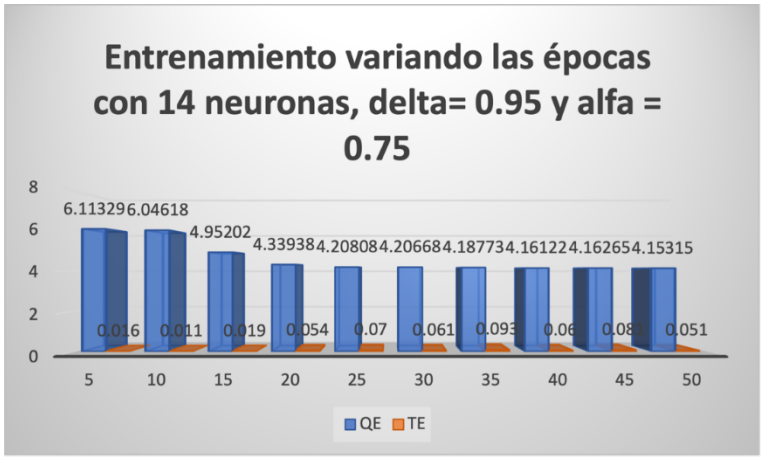

Figura 5.5 Entrenamiento SOM serial variando las épocas con 14x14 neuronas.

Finalmente, el análisis de tiempos para todas las épocas de entrenamiento con redes de 10x10, 12x12 y 14x14 neuronas es presentado en la Figura 5.6, donde se muestra que el aumento del número de épocas en el entrenamiento se da de forma lineal. Pero, con el aumento en el número de neuronas de la red parece tener un incremento de tiempo cuadrático, consistente con el aumento de cuadrático del número de neuronas.

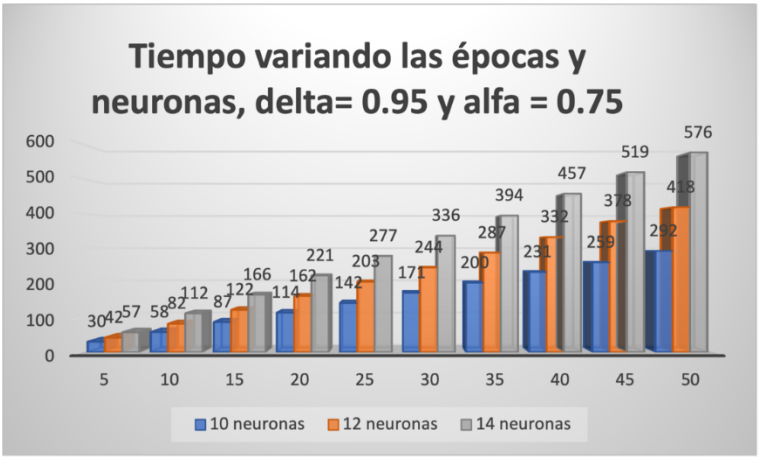

Figura 5.6 Tiempo de entrenamiento SOM serial para redes con 10, 12 y 14 neuronas.

De acuerdo a las mediciones de tiempo variando el número de épocas y el número de neuronas de la matriz de pesos, se puede observar el tiempo aumenta en una forma lineal y exponencial respectivamente. Esto muestra la necesidad de paralelización del algoritmo en base a una granularidad gruesa abordada en el diseño del algoritmo propuesto.

6. Conclusiones

Se implementó completamente el código para el entrenamiento secuencial del mapa autoorganizado (*SOM*), realizando cada una de las etapas principales del entrenamiento de la red como lo son el cálculo de la distancia, la obtención de la neurona ganadora, el cálculo de la vecindad y la actualización de la red completa. El programa completo es mostrado en el anexo A.

Para analizar el desempeño del algoritmo, se programaron las métricas de desempeño del error de cuantización y el error topológico, donde el primero describe que tan similar es la neurona ganadora con cada entrada de datos, mientras que el error topológico indica que tan bien agrupadas están la neurona ganadora con la segunda neurona más cercana con el mismo dato de entrada.

Realizando múltiples pruebas de entrenamiento de la red *SOM* sobre el conjunto de entrenamiento *MNIST* variando los parámetros de entrenamientos como α, δ, las épocas y las neuronas, con el objetivo de encontrar los mejores desempeños de acuerdo con los hiper-parámetros, consiguiendo una relación costo-beneficio entre los parámetros de la red, la épocas y tamaño de la red. Concluyendo que una red de 14 neuronas con 30 épocas y un parámetro $\alpha = 0.75$ y $\delta = 0.95$, se obtiene un buen desempeño sin sobre entrenar. Esto contemplando que el error de cuantización total $QE = 4.2$ y error topológico $TE = 0.061$, lo que significa que una distancia promedio de 0.0042 entre la neurona ganadora y la imagen de dígito de entrada y que sólo 61 de las 1000 entradas no tuvieron una segunda neurona ganadora como vecina.

Además de encontrar la combinación de hiper-parámetros, de las pruebas de entrenamiento con el fin de establecer una mejora en el proceso de entrenamiento se pueden concluir que la selección de alfa y delta no es crítica para el desempeño de la *SOM*, sin embargo, debe mantenerse en un rango alto para mantener su convergencia. Es decir, solo se requiere la selección de una única combinación alfa y delta pare entrenar la red. Con respecto a la selección del número de neuronas no es un parámetro crítico para el desempeño de la *SOM*, depende más de la complejidad de los datos que definirán el tamaño, lo que requiere realizar entrenamiento con un mapa de tamaño mínimo y aumentarlo hasta generar el desempeño deseado sin aumentar en dos veces el mapa. Entre más grande el mapa de neuronas mejora el rendimiento de la SOM, pero este decremento es relativamente poco en comparación con el incremento al costo computacional. El incremento computacional por el aumento de épocas de entrenamiento muestra un comportamiento lineal, mientras que el incremento computacional por el aumento del tamaño del mapa de neuronas.

El diseño del entrenamiento paralelo maestro – trabajadores propuesto tiene como objetivo realizar una división de datos dentro de los trabajadores a través de instrucciones de *MPI*, donde cada trabajador realizará el entrenamiento serial de su red *SOM* con los datos asignados y al final el maestro recopila todas las redes *SOM* de los trabajadores y entrena una sola red *SOM* global con las neuronas de las redes *SOM* recolectadas. Esto debe permitir que la *SOM* global mantenga la topología de los datos y tenga se realice un entrenamiento distribuido.

La programación del diseño del entrenamiento paralelo propuesto se realizó en una forma preliminar (ver anexo B), empleando vectores definidos, siguiendo los diagramas de flujo mostrados en las secciones 4.1 y 4.2, el programa funciona conceptualmente de acuerdo

a la propuesta. El programa del entrenamiento paralelo no es parte de este trabajo de tesina y queda como trabajo futuro con el fin de obtener la viabilidad y el rendimiento del diseño propuesto.

Bibliografía

[1] Kohonen, T., & Maps, S. O. (1995). Springer-Verlag.
[2] https://scholar.google.com.mx/scholar?hl=es&as_sdt=0%2C5&q=SOM+kohonen&btnG=
[3] T. Kohonen, The self-organizing maps, Proc. IEEE 78 (1990) 1464–1480.
[4] M. Ceccarelli, A. Petrosino, R. Vaccaio, Competitive neural networks on message passing parallel computers, Concurrency: Practice and Experience 5 (1993).
[5] S. Kaski, J. Kangas, T. Kohonen, Bibliography of self organizing map (SOM) papers: 1981–1997, Neural Computing Survey 1 (1) (1998).
[6] Documentación MPI https://www.mpi-forum.org/docs/
[7] Interfaz de paso de mensajes https://es.wikipedia.org/wiki/Interfaz_de_Paso_de_Mensajes.
[8] D.G. Martínez y S. Rodríguez, Instalación de OpenMPI, Universidad de Granada. España 2010
[9] M. Oja, S. Kaski, T. Kohonen, Bibliography of self organizing map (SOM) papers: 1998–2001 Addendum, Neural Computing Survey 3 (1) (2003) 2003.
[10] C. Bishop, M. Svensen, Developments of the generative topographic mapping, Neurocomputing 21 (1998) 203–224.
[11] M. Pena, W. Barbakh, Topology-preserving mappings for data visualisation, in: Principal Manifolds for Data Visualization and Dimension Reduction, in: Lecture Notes in Computational Science and Engineering, vol. 58, 2008, pp. 131–150. doi:10.1007/978-3-540-73750-6_5.
[12] C. Garcia, M. Prieto, A. Pascual-Montano, A speculative parallel algorithm for self organizing maps, in: Proceedings of the International Conference ParCo Parallel Computing: Current & Future Issue of High-End Computing, 2005.
[13] M. Cottrel, J.C. Fort, P. Letremy, Advantages and drawbacks of the batch Kohonen algorithm, in: 10th European Symp. On Artificial Neural Network, Bruges, Belgium, 2005.
[14] H. Jin, M.L. Wong, K.S. Leung, W. Shum, K.-S. Leung, Expanding self-organizing map for data visualization and cluster analysis, Information Sciences 163 (1–3) (2004).
[15] M. Morchren, A. Ultsch, ESOM Maps, Technical Report 45, Dept. CS, University of Marburg, Germany, 2005.
[16] D. Alahakoon, S.H. Halgamuge, B. Srinivasan, Dynamic self-organizing maps with controlled growth for knowledge discovery, IEEE Transactions on Neural Networks 11 (3) (2000).
[17] A. Faro, D. Giordano, F. Maiorana, Discovering complex regularities by adaptive self organizing classification, in: The Second World Enformatika Conference, WEC'05, Instanbul, 2005.
[18] T. Ayadi, T.M. Hamdani, I.E.E.E. Member, M. Adel, A.M. Alimi, Enhanced data topology preservation with multilevel interior growing self-organizing maps, in: Proc. World Congress on Engineering, WCE 2010, London, 2010.
[19] G. Denny, J. Williams, P. Christen, ReDSOM: relative density visualization of temporal changes in cluster structures using self-organizing maps, in: Proceedings of the Eighth IEEE International Conference on Data Mining, ICDM, 2008, pp. 173–182.
[20] Faro, A., Giordano, D., & Maiorana, F. (2011). Mining massive datasets by an unsupervised parallel clustering on a GRID: Novel algorithms and case study. *Future Generation Computer Systems*, *27*(6), 711-724.
[21] Gajdoš, P., & Moravec, P. (2010, April). Two-step modified SOM for parallel calculation. In *Proceedings of DATESO* (pp. 13-21).
[22] OpenMP http://www.openmp.org/
[23] Radenski, Atanas. "Shared memory, message passing, and hybrid merge sorts for standalone and clustered smps." Proc. PDPTA vol. 11, pp. 367-373, 2011.
[24] Patel, B., Tibrewal, Y., Jajoo, A., & Joshi, A. (2015). An Efficient Parallel Algorithm for Self-Organizing Maps using MPI-OpenMP based Cluster. International Journal of Computer Applications, 975, 8887.

7. Anexos

7.1. Anexo A: train_SOM_sequential.cpp

```
/* file: train_SOM_sequential : Este archivo contiene la función "main".
La ejecución del programa comienza y termina ahí.
    @author David Anguiano
    version 1.0

    compiler: g++ file.cpp -o file -lm
    usage: .\file -i file.csv -c featuresRow -r classesCol -p col*row
Weights epochs alpha delta decayLrn
*/

#include <iostream>
#include <math.h>
#include <fstream>
#include <string>
#include <sstream>
#include <iomanip>
#include <cstring>
#include <limits.h>
#include <vector>
#include <time.h>
#include <chrono>
#include <sys/time.h>
#include <ctime>

using std::cout;
using std::string;
using vec = std::vector<float>;
using vecStr = std::vector<string>;
using matrix = std::vector<vec>;
using matrixStr = std::vector<vecStr>;
using weight = std::vector<matrix>;
matrix MinMax;
matrix X;
vecStr Rc;
vecStr Rh;
matrix ut;
matrixStr out;
vec emp;

int main(int argc, char const *argv[])
{
    int headerData;
    int classData;
    string pathData;

    // Parametros del algoritmo
    int E;        // Epocas de repeticion
    int N;        // Numero de neurona
    float Delta0; // Ancho de la vecindad inicial
```

```
    float Alfa0;  // Taza de aprendizaje inicial
    char Dlr;     // Tipo de decaimiento de la taza de aprendizaje

    cout << "USAR: RED -i file.csv -c featuresRow -r classesCol -p
col*row Weights epochs alpha delta decayLrn \n";

    time_t startTime = time(nullptr);
    // 1. Lee los hiperparametros desde la linea de comandos ------------
-
    for (int i = 0; i < argc; i++)
    {
        if (!strcmp(argv[i], "-i"))
        {
            pathData = argv[++i];
        }
        else if (!strcmp(argv[i], "-c"))
        {
            headerData = atoi(argv[++i]);
        } // define Cells size 'atoi-ascii to int'
        else if (!strcmp(argv[i], "-r"))
        {
            classData = atoi(argv[++i]);
        }
        else if (!strcmp(argv[i], "-p"))
        {
            N = atoi(argv[++i]);
            E = atoi(argv[++i]);
            Alfa0 = atof(argv[++i]);
            Delta0 = atof(argv[++i]);
            string temp = argv[++i];
            Dlr = temp[0];
        }
    }

    /*Lee e imprime el vector de entrada, renglon = dimension,
columna=muestras*/
    leerCSV(pathData, headerData, classData);
    MinMax = normaliza(X, 0, 255); // X,0,250 para imagenes
    // mostrarMat(X, 1);
    ////////////// Guardar min y max y etiquetar neuronas ganadora,
obtener la utilización de neuron
    int m = (int)X[0].size();
    int T = (int)X.size();
    // int T = 10;
    cout << "m: " << m << " T: " << T << "\n";
    weight W; // tamaño W[N][N][m]
    matrix d; // tamaño d[N][N]
    matrix H; // tamaño H[N][N]

    float alfa = Alfa0;
    float delta = Delta0;

    vec Error;
    float EQ = 0;
    float TQ = 0;

    /*Inicio aleatorio de las neuronas*/
```

```
    // std::cout << "Matriz de pesos inicial" << std::endl;
    W = iniciarW(N, m);
    // mostrarRed(W);
    /*Inicia entrenamiento por epocas*/
    for (int p = 1; p <= E; p++) // numero de epocas por repeticion
    {
        //  std::cout <<
"==============================================";
        // std::cout << "Epoca = " << p << "\n";

        // Actualización secuencial
        for (int i = 0; i < T; i++) // numero muestras
        {
            /* Normaliza el vector de entrada */
            vec Y = vecNorm(MinMax, X[i], 1);

            /*Calculo de la distancia*/
            // cout << "Matriz de distancias \n";
            matrix d = distancia(W, Y);
            // mostrarMat(d,0);

            /*Calcula la menor distancia (neurona ganadora)*/
            vec BMU = ganadora(d);
            // cout << "BMU = {" << BMU[0] << ", " << BMU[1] << "} = " <<
BMU[2] << std::endl;
            //  cout << "SMU = {" << BMU[3] << ", " << BMU[4] << "} = "
<< BMU[5] << std::endl;

            /*Calcula las vecindades*/
            matrix H = vecindad(BMU, N, delta);
            // cout << "Matriz de vecindades \n";
            // mostrarMat(H,0);

            /*Actualizacion de las neuronas*/
            W = actualiza(W, H, Y, alfa);
        }
        //  Calcular metricas de topología
        Error = errorMeasure(W);
        cout << "Ep:" << p << " QE: " << Error[0] << " TE: " << Error[1]
<< " alfa: " << alfa << " delta: " << delta << "\n";
        //  update delta, alfa
        alfa = decayFcn(Alfa0, p, E, Dlr);
        delta = decayFcn(Delta0, p, E, 'E');
        // cout << "p: " << p << " alfa: " << alfa << " delta: " << delta
<< std::endl;
    }
    // cout << "matriz final \n";
    // mostrarRed(W);
    cout << "\n\n matriz de utilizacion";
    mostrarMat(ut, 0);
    cout << "\n\n matriz de clases";
    mostrarStr(out, 0);
    // cout << "Ep:" << p << " QE: " << Error[0] << " TE: " << Error[1]
<< " alfa: " << alfa << " delta: " << delta << "\n";
    time_t endTime = time(nullptr);
    float tFinal = endTime - startTime;
    cout << "tiempo seg: " << tFinal << " min: " << tFinal / 60 << "\n";
```

```
    weight Wd = denormaliza(W, MinMax);
//  mostrarRed(W);
//  mostrarRed(Wd);
//  guardarPesos(W, pathData, E, N, Delta0, Alfa0, Dlr, Error[0],
Error[1]);
    guardarPesos(Wd, pathData, E, N, Delta0, Alfa0, Dlr, Error[0],
Error[1]);
    return 0;
}

void leerCSV(const string &rutaArchivo, int H, int C)
{
    char delim = ',';
    matrix result;
    string row, item;

    std::ifstream in(rutaArchivo);
    int countL = 0;
    while (std::getline(in, row))
    {
        if (countL == H)
        {
            std::stringstream ss(row);
            while (getline(ss, item, delim))
            {
                Rh.push_back(item);
            }
            countL++;
            cout << "Header: " << Rh.size() << " "
                 << "\n";
        }
        else
        {
            int countC = 0;
            vec R;
            std::stringstream ss(row);
            while (getline(ss, item, delim))
            {
                if (countC == C)
                {
                    Rc.push_back(item);
                    countC++;
                }
                else
                {
                    R.push_back(stof(item));
                    countC++;
                }
                countL++;
            }
            X.push_back(R);
        }
    }
    cout << "Classes: " << Rc.size() << " "
         << "\n";
    in.close();
    // return result;
```

```
}

void guardarPesos(const weight &W, string path, int E, int N, float D,
float A, float Dlr, float QE, float TE)
{
    char delim = ',';
    matrix result;
    string row, item;
    std::stringstream nombre;
    // std::ofstream archivo;
    nombre << "SOM-data-" << path << "-E-" << E << "-N-" << N << "-D-"
           << D << "-A-" << A << "-Dec-" << Dlr << "-QE-" << QE << "-TE-"
<< TE << ".csv";
    cout << nombre.str() << "\n";
    // archivo.open(nombre.str(),std::fstream::out);
    std::ofstream archivo(nombre.str());
    if (!archivo.is_open())
    {
        cout << "Error al abrir el archivo\n";
        exit(EXIT_FAILURE);
    }

    for (const matrix &col : W)
    {
        for (const vec &row : col)
        {
            int c = 0;
            for (const float &s : row)
                if (c == 0)
                {
                    archivo << s;
                    c = 1;
                }
                else
                {
                    archivo << "," << s;
                }
            archivo << std::endl;
        }
        archivo << std::endl;
    }

    archivo.close();
    cout << "Escrito correctamente \n";
}

void mostrarMat(const matrix &M, int show)
{
    cout << "\n \n
////////////////////////////////////////////////////////////////////////
\n";
    if (show == 1)
    {
        for (const string &head : Rh)
            cout << std::setw(3) << std::left << head << " ";
        cout << std::endl;
    }
```

```
    int count = 0;
    for (const vec &row : M)
    {
        for (const float &s : row)
            cout << std::setw(3) << std::left << s << " ";
        if (show == 1)
        {
            cout << Rc[count];
            count++;
        }
        cout << std::endl;
    }
}

void mostrarStr(const matrixStr &M, int show)
{
    cout << "\n \n
////////////////////////////////////////////////////////////////////////////
\n";
    if (show == 1)
    {
        for (const string &head : Rh)
            cout << std::setw(3) << std::left << head << " ";
        cout << std::endl;
    }

    int count = 0;
    for (const vecStr &row : M)
    {
        for (const string &s : row)
            cout << std::setw(3) << std::left << s << " ";
        cout << std::endl;
        // getchar();
    }
}

void mostrarRed(const weight &W)
{
    cout << "\n \n
////////////////////////////////////////////////////////////////////////////
\n";
    cout << "Matriz de pesos" << std::endl;
    for (const matrix &col : W)
    {
        for (const vec &row : col)
        {
            cout << "{ ";
            for (const float &s : row)
                cout << s << " ";
            cout << "}    ";
        }
        cout << std::endl;
    }
}

weight iniciarW(int N, int m)
```

```
{
    weight W;
    srand(time(NULL));

    for (int j = 0; j < N; j++)
    {
        matrix Mat;
        for (int k = 0; k < N; k++)
        {
            vec Row;
            for (int l = 0; l < m; l++)
            {
                Row.push_back(0.1 * (rand() % 11));
            }
            Mat.push_back(Row);
        }
        W.push_back(Mat);
    }
    return W;
}

matrix normaliza(const matrix &X, int setMin, int setMax)
{
    matrix norm;
    vec min;
    vec max;
    int m = X.size();
    int n = X[0].size();
    if (setMin != setMax)
    {
        for (int k = 0; k < n; k++)
        {
            min.push_back(setMin);
            max.push_back(setMax);
            //      cout << "k:" << k << " max: " << max[k] << " min: "
<< min[k] << "\n";
        }
    }
    else
    {
        for (int k = 0; k < n; k++)
        {
            min.push_back(X[0][k]);
            max.push_back(X[0][k]);
            //      cout << "k:" << k << " max: " << max[k] << " min: "
<< min[k] << "\n";
        }
        for (int i = 1; i < m; i++)
        {
            for (int j = 0; j < n; j++)
            {
                if (min[j] > X[i][j])
                {
                    min[j] = X[i][j];
                }
                if (max[j] < X[i][j])
                {
```

```
                    max[j] = X[i][j];
                }
            }
        }
    }

    for (int k = 0; k < n; k++)
    {
        //cout << "k:" << k << " max: " << max[k] << " min: " << min[k]
<< "\n";
    }

    norm.push_back(min);
    norm.push_back(max);
    return norm;
}

weight denormaliza(const weight &W, const matrix norm)
{
    weight denorm;
    vec min = norm[0];
    vec max = norm[1];
    int m = X.size();
    int n = X[0].size();

    for (int j = 0; j < W.size(); j++) // recorre las columnas de la SOM
    {
        matrix tmp1;
        for (int k = 0; k < W[j].size(); k++) // recorre los reglones de
la SOM
        {
            vec tmp2;
            for (int s = 0; s < W[j][k].size(); s++) // recorre los
atributos
            {

                tmp2.push_back((W[j][k][s] * (max[s] - min[s]) +
min[s]));
            }
            tmp1.push_back(tmp2);
        }
        denorm.push_back(tmp1);
    }
    return denorm;
}

vec vecNorm(const matrix &N, const vec &X, int pixel)
{
    vec temp;
    for (int i = 0; i < X.size(); i++)
    {
        float temp2;
        if (pixel == 1)
        {
            temp2 = X[i] / 255;
            temp.push_back(temp2);
            // cout << temp2 << std::endl;
```

```
        }
        else
        {
            if (N[1][i] > N[0][i])
            {
                temp2 = (X[i] - N[0][i]) / (N[1][i] - N[0][i]);
                temp.push_back(temp2);
            }
        }
    }
    return temp;
}

matrix distancia(const weight &W, const vec &x)
{
    matrix dist;
    for (int j = 0; j < W.size(); j++) // tamaño de la red neuronal en
vertical y horizontal.
    {
        vec row;
        for (int k = 0; k < W[j].size(); k++)
        {
            float res = 0;                              // es la suma
            for (int l = 0; l < W[j][k].size(); l++) // la dimension del
vector de entrada
            {
                res += pow((x[l] - W[j][k][l]), 2); // distancia
euclidiana
            }
            res = sqrt(res);
            row.push_back(res); // la distancia resultante por neurona
        }
        dist.push_back(row);
    }
    return dist;
}

vec ganadora(const matrix &dist)
{
    vec BMU;
    float r = dist[0][0];
    int a = 0;
    int b = 0;
    int a2 = 0;
    int b2 = 0;
    float r2 = 0;
    for (int j = 0; j < dist.size(); j++)
    {
        for (int k = 0; k < dist[0].size(); k++)
        {
            if (dist[j][k] < r)
            {
                a2 = a;
                b2 = b;
                r2 = r;
                b = k;
                a = j;
```

```
                r = dist[j][k];
            }
        }
    }
    BMU.push_back(a);  // ren
    BMU.push_back(b);  // col
    BMU.push_back(r);  // neurona ganadora
    BMU.push_back(a2); // ren2
    BMU.push_back(b2); // col2
    BMU.push_back(r2); // segunda neurona ganadora

    return BMU;
}

matrix vecindad(const vec &BMU, int N, float delta)
{
    /*Actualizacion de las neuronas*/
    float r;
    matrix H;
    for (int j = 0; j < N; j++)
    {
        vec row;
        for (int k = 0; k < N; k++)
        {
            r = exp((-sqrt(pow((BMU[0] - j), 2) + pow((BMU[1] - k), 2)))
/ (2 * pow(delta, 2)));
            row.push_back(r);
        }
        // cout << "columna" << row.size() << "\n";
        H.push_back(row);
    }
    // cout << "renglon" << H.size() << "\n";
    return H;
}

weight actualiza(weight &W, matrix &H, vec &X, float alfa)
{
    for (int j = 0; j < W.size(); j++) // recorre las columnas de la SOM
    {
        for (int k = 0; k < W[j].size(); k++) // recorre los reglones de
la SOM
        {
            for (int l = 0; l < W[j][k].size(); l++) // recorre los
atributos
            {
                float r = W[j][k][l] + alfa * H[j][k] * (X[l] -
W[j][k][l]);
                if (!isnan(r))
                {
                    W[j][k][l] = r;
                }
            }
        }
    }
    return W;
}
```

```
float decayFcn(float val, int p, int E, int lr)
{
    float Valor;
    float temp;
    switch (lr)
    {
    case 'L': // lineal
        temp = (float)(p / E);
        Valor = val * (1 - temp);
        break;
    case 'I': // inverso
        temp = (float)(1 / p);
        Valor = val * temp;
        break;
    case 'E': // exponencial
        temp = (float)p / E;
        Valor = val * exp(-2*temp);
        break;
    default:
        temp = (float)p / E;
        Valor = val * (1 - temp);
    }
    return Valor;
}

vec errorMeasure(const weight &W)
{
    // cout << "1 \n";
    float QE = 0;
    float TE = 0;
    matrix temp;
    matrixStr tempO;

    //  out.clear();
    for (int i = 0; i < W.size(); i++)
    {
        vec tmp;
        vecStr tmp2;
        for (int j = 0; j < W[i].size(); j++)
        {
            tmp.push_back(0);
            tmp2.push_back("|");
        }
        temp.push_back(tmp);
        tempO.push_back(tmp2);
    }

    for (int i = 0; i < X.size(); i++) // numero de muestras
    {
        vec Y = vecNorm(MinMax, X[i], 1);
        /*Calculo de la distancia*/
        matrix d = distancia(W, Y);

        /*Calcula la menor distancia (neurona ganadora)*/
        vec BMU = ganadora(d);
        temp[BMU[0]][BMU[1]] += 1;
```

```
        // cout << "2 Rc: " << Rc[i].size() << " X: " << X.size() <<
"\n";
        string temp3;
        temp3.push_back(Rc[i].at(0));
        // cout << "3 \n";

        // cout << "ganadora: " << BMU[0] << " " << BMU[1] << " class: "
<< temp3 << std::endl;
        tempO[BMU[0]][BMU[1]] = tempO[BMU[0]][BMU[1]] + temp3;
        QE += BMU[2];
        if ((BMU[0] - BMU[3]) > 1 && (BMU[1] - BMU[4]) > 1)
        {
            TE += 1;
        }
    }
    vec E;
    QE = QE / X.size();
    TE = TE / X.size();
    E.push_back(QE);
    E.push_back(TE);
    ut = temp;
    out = tempO;
    return E;
}
```

7.2. Anexo B: train_SOM_parallel

```
// train_SOM_parallel.cpp : Este archivo contiene la función "main". La
ejecución del programa comienza y termina ahí.
//
//cabeceras
#include <iostream>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <mpi.h>


/*Ejemplo de clasificación del Iris de una flor. El último es el tipo de
iris
6.3,2.5,5.0,1.9,Iris-virginica
6.5,3.0,5.2,2.0,Iris-virginica
6.2,3.4,5.4,2.3,Iris-virginica
5.9,3.0,5.1,1.8,Iris-virginica
6.1,3.0,4.6,1.4,Iris-versicolor
5.8,2.6,4.0,1.2,Iris-versicolor
5.0,2.3,3.3,1.0,Iris-versicolor
5.6,2.7,4.2,1.3,Iris-versicolor
5.0,3.2,1.2,0.2,Iris-setosa
5.5,3.5,1.3,0.2,Iris-setosa
4.9,3.1,1.5,0.1,Iris-setosa
4.4,3.0,1.3,0.2,Iris-setosa*/
//Variables globales
//Aqui definimos el punto 1 del algoritmo serial SOM
```

```
    int E;//Epocas
    int N;//valor de referencia de las "W"
    int n;//es el numero de dimensiones de las muestras
    int T; //Define el tamaño de las muestras
    float delta;//
    float alfa;//

    float * x = NULL;//Muestras de entrada.
    float * h = NULL;//Matriz de vecindades
    float * d = NULL;//Matriz de distancias
    float * w = NULL;//Matriz de pesos
    float * recv_X = NULL;
    float * recv_Y = NULL;

    float r = 0; //contener una variable temporal
    float BMU = 0;//Neurona ganadora
    int a = 0;//almacena el indice j
    int b = 0;//almacena el valor de k
    int p = 0;//indica el valor que recorre a epocas
    int i = 0;//indice horizontal de la matriz de entrada "X".
    int j = 0;//indice vertical de la matriz(renglon de la matriz de
pesos)
    int k = 0;//indica el valor que obtiene de la matriz(columna de la
matriz de pesos)
    int l = 0;//indice de la dimension de la muestra(profundidad de la
matriz de pesos)

int main(int argc, char const* argv[])
{

    //NOTA: Agregar lectura de argumentos
     //Nota: poner variables de mpi
    int size,rank;

    if (argc!=3){
        cout << "USAR: RED2 Ruta_del_archivo Archivo_de_salida \n";
        exit(1);
        }

    const char *inFileName = argv[1];
    const char *outPrefix = argv[2];

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);


if (rank == 0)
    {
     x = readMatrix(inFileName,T,n);
     Y = readMatrix(inFileName,T,n);
    //Aqui definimos el punto 1 del algoritmo serial SOM
     E = 100;//Epocas
     N = 4;//valor de referencia de las "W"
     //n = 2;//es el numero de dimensiones de las muestras
     //T = 5; //Define el tamaño de las muestras
     delta = 0.5;//
```

```
     alfa = 0.7;//
     int kappa=N*N*n;//contador de elementos de la matriz de pesos "w".
     w = (float*)malloc(kappa*sizeof(float*));
     //x[5][2] = { {1,2},{3,4},{5,6},{7,8},{9,10} };//Muestras de
entrada.
     recv_Y=(float*)malloc(mu*sizeof(float*));//Aqui va la matriz que
recibe Y.
/*Imprime el vector de entrada, renglon = dimension, columna=muestras*/

    std::cout << "X=" << std::endl;
    for (i = 0; i < T; i++)
    {
        for (l = 0; l < n; l++)
        {
            std::cout << x[i*n+l];
            std::cout << ", ";
        }
        std::cout << std::endl;
    }
    std::cout << std::endl << std::endl;
/*EMPIEZA EL ENTRENAMIENTO ALGORITMO*/
/*Punto 1 inicializar la matriz de pesos(NxNxn) con valores aleatorios
entre 0 y 1, de las neuronas*/

    std::cout <<"Inicial W=" << std::endl;
    for (j = 0; j < N; j++)
    {
        for (k = 0; k < N; k++)
        {
            for (l = 0; l < n; l++)
            {
                w[j*N*n+k*n+l] = (rand()%11);
                std::cout << w[j*N*n+k*n+l] ;
                std::cout << ", ";
            }
            std::cout << "   ";
        }
        std::cout << std::endl;
    }
    std::cout << std::endl << std::endl;
    //Aqui va el Bcast: ENVIA W EL NODO MAESTRO, (ENVIA UNA VARIABLE
COMPLETA, MATRIZ, DATOS)


  }
   MPI_Barrier(MPI_COMM_WORLD);
   MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);
   MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
   MPI_Bcast(&T, 1, MPI_INT, 0, MPI_COMM_WORLD);
     h = (float*)malloc(N*N*sizeof(float*));//Matriz de vecindades
     d = (float*)malloc(N*N*sizeof(float*));//Matriz de distancias

     int kappa=N*N*n;//contador de elementos de la matriz de pesos "w".
     //w[4][4][2];//Matriz de pesos

 MPI_Bcast(&w, kappa, MPI_FLOAT, 0, MPI_COMM_WORLD);
 MPI_Bcast(&E, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

```
 MPI_Bcast(&delta, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
 MPI_Bcast(&alfa, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);



     //AQUI VA EL SCATTER: ENVIA MATRIZ "X" O MUESTRAS SE DIVIDIRA POR
MUESTRAS POR DIMENSION 1 (ENVIA TROZOS DE INFORMACION A LOS WORKERS)
     int mu=ceil(T/(size*1.0))*n;
     recv_X=(float*)malloc(mu*sizeof(float*)); //asigna memoria local
dinamica
     MPI_Scatter(&x, mu, MPI_FLOAT, recv_X, mu, MPI_FLOAT, 0,
MPI_COMM_WORLD)
    /*Punto 2 Repite el entrenamiento E veces */

    //Esto lo ejecutan todos los nodos
    for (p = 1; p <= E; p++)//numero de epocas por repeticion
    {
//Punto 3 recorre las muestras de la matriz de entrada "X"
        for (i = 0; i < mu; i++)//numero muestras recorre la fila de
muestras. recorre el
        {

            std::cout <<
"=============================================";
            std::cout << "Epoca = " << p << ", Muestra = " << i << ", d="
<< std::endl;
//Punto 4 Calcula la distancia euclidiana de la matriz X_i y W_jk para
todas las neuronas
            for (j = 0; j < N; j++)//tamaño de la red neuronal en
vertical y horizontal.
            {
                for (k = 0; k < N; k++)
                {
                    float r = 0;// es la sumatoria de diferencia de los
cuadrados
                    for (l = 0; l < n; l++)//la dimension del vector de
entrada
                    {
                        r += pow((recv_X[i*n+l] - w[j*N*n+k*n+l]),2);
//distancia euclidiana suma el valor anterior con el nuevo calculo
                        //std::cout << "l =" << l << ", r =" << r << "
";
                    }
                    r = sqrt(r);
                    d[j*N+k] = r;//la distancia resultante por neurona
                    std::cout << d[j*N+k];
                    std::cout << "  ";
                }
            std::cout << std::endl;
            }
            std::cout << std::endl << std::endl;

/* Punto 5 Calcula la neurona ganadora con la menor distancia* la matriz
de distancias */
            r = d[0];
            for (j = 0; j < N; j++)
            {
```

```
                for (k = 0; k < N; k++)
                {
                    if (d[j*N+k] < r)
                    {
                        b = k;
                        a = j;
                        r = d[j*N+k];
                    }
                }
            }
            BMU = r;//neurona ganadora
            std::cout << "Ganadora = [" << a << "][" << b << "] = " <<
BMU; //se guardan las coordenadas en "a" y "b"
            std::cout << std::endl << std::endl;

/*Actualizacion de las neuronas*/


            std::cout << "Actualizacion W=" << std::endl;
// Punto 6 empieza el conjunto de vecindad de la coordenada BMU (a,b)
contra todas las neuronas (j,K).
            for (j = 0; j < N; j++)
            {
                for (k = 0; k < N; k++)
                {
                    r= exp((-sqrt(pow((a - j),2) + pow((b - k),2))) /
(2*pow(delta,2)));//Calculando la magnitud del vector d
                    h[j*N+k] = r;//Fin del calculo de la vecindad.
                    //std::cout << "vecindad h=" << h[j][k]<< ", ";
                    //r = 0;
//Punto 7 actualizar la matriz de pesos de la BMU y sus vecindades.
                    for (l = 0; l < n; l++)
                    {
                        r = w[j*N*n+k*n+l];
                        r = r + alfa * h[j*N+k]*(x[i*N+l] - r);
                        w[j*N*n+k*n+l] = r;
                        std::cout << w[j*N*n+k*n+l];
                        std::cout << ", ";
                    }
                    std::cout << "   ";
                }
                std::cout << std::endl;
            }
            std::cout << std::endl << std::endl;
        }//Punto 8 termina una epoca con todas las muestras
    /*Ahi van las actualizaciones de delta y alfa*/
    }
    //Aqui va el gather
    recv_Y=(float*)malloc(*sizeof(float*));
    MPI_Gather(&w, kappa, MPI_FLOAT, recv_Y, kappa, MPI_FLOAT, 0,
MPI_COMM_WORLD);

//Definir el tamaño de recv_Y con malloc. kappa * size(float)
//Solo el maestro debe calcular una som con recv_Y como entrada.

}
```